MAQAO
Hands-on exercises

Profiling with ONE View 1

Optimising a code

Profiling with ONE View 2 & 3

- Login to the cluster with X11 forwarding

```
> ssh -Y -o ProxyCommand='ssh <username>@hpc.cenaero.be \
nc zenobe 22' <username>@zenobe
```

- Copy handson material to your HOME directory

```
> cd $HOME
> tar xf /projects/uvsq/PRACET1FWB_MAQAO/tutorial/MAQAO_HANDSON.tgz
```

- Load MAQAO environment

```
> module load parallel_tools/maqao/2.8.6
```

- Load Intel environment (or other)

```
> module load compiler/intel/2018.0.1 intelmpi/5.0.3.049/64
```

- Optional: mount directories remotely on local machine (if using Linux)

```
> sudo apt install sshfs
> mkdir home_remote
> sshfs -o ProxyCommand='ssh <username>@hpc.cenaero.be \
nc zenobe 22' <username>@zenobe: home_remote
```

Compiling NAS Parallel Benchmarks

- Copy NAS sources to your working directory

```
> cd $HOME
> tar -xvf /projects/uvsq/PRACET1FWB_MAQAO/tutorial/NPB3.4-MZ.tgz
```

- Compile NAS suite

```
> cd $HOME/NPB3.4-MZ/NPB3.4-MZ-MPI/
> module load compiler/intel/2018.0.1 intelmpi/5.0.3.049/64
> make suite
```

- (Optional) To execute a sample run of a NAS:

```
> cd $HOME/NPB3.4-MZ/NPB3.4-MZ-MPI/
> vim job_npb.pbs
> qsub job_npb.pbs
```

Profiling with MAQAO ONE View 1

Cédric Valensi

Setup ONE View for batch mode

- The ONE View configuration file must contain all variables for executing the application.
- Retrieve the configuration file prepared for the NAS in batch mode from the MAQAO_HANDSON directory

```
> cd $HOME/NPB3.4-MZ/NPB3.4-MZ-MPI/
> cp $HOME/MAQAO_HANDSON/npb/config_maqao_npb.lua .
> less config_maqao_npb.lua
```

```
binary = "bin/bt-mz.C.x"
...
batch_script = "job_npb_maqao.pbs"
...
batch_command = "qsub <npb_script>"
...
number_nodes = 4
...
number_tasks_nodes = 24
...
number_processes = 4
...
mpi_command = "mpirun"
...
omp_num_threads = 6
...
```

Review jobscript for use with ONE View

- All variables in the jobscript defined in the configuration file must be replaced with their name from it.
- Retrieve jobscript modified for ONE View from the MAQAO_HANDSON directory.

```
> cd $HOME/NPB3.4-MZ/NPB3.4-MZ-MPI/
> cp $HOME/MAQAO_HANDSON/npb/job_npb_maqao.pbs .
> less job_npb_maqao.pbs
```

```
...
#PBS -l select=4<number_nodes>:ncpus=24<number_tasks_nodes>:
mem=5gb:mpiprocs=4<number_processes>:
ompthreads=6<omp_num_threads>
...
export OMP_NUM_THREADS=6<omp_num_threads>
mpirun bin/bt-mz.C.x
<mpi_command> <run_command>
...
```

Launch MAQAO ONE View

- Launch ONE View

```
> cd $HOME/NPB3.4-MZ/NPB3.4-MZ-MPI/
> maqao oneview --create-report=one \
--config=config_maqao_npb.lua -xp=maqao_npb
```

- The -xp parameter allows to set the path to the experiment directory, where ONE View stores the analysis results and where the reports will be generated.

- If -xp is omitted, the experiment directory will be named maqao_<timestamp>.

- WARNING: If the directory specified with -xp already exists, ONE View will reuse its content but not overwrite it.

Display MAQAO ONE View results

- The HTML files are located in <exp-dir>/RESULTS/<binary>_one_html, where <exp-dir> is the path of he experiment directory (set with -xp) and <binary> the name of the executable.

```
> firefox maqao_npb/RESULTS/bt-mz.C.x_one_html/index.html
```

- It is also possible to compress and download the results to display them:

```
> tar -zcf $HOME/bt_html.tgz maqao_npb/RESULTS/bt-mz.C.x_one_html
```

```
> scp -o ProxyCommand='ssh <username>@hpc.cenaero.be \
nc zenobe 22' <username>@zenobe:bt_html.tgz .
> tar xf bt_html.tgz
> firefox maqao_npb/RESULTS/bt-mz.C.x_one_html/index.html
```

- A sample result directory is in MAQAO_HANDSON/npb/bt-mz.C.x_one_html/
- Results can also be viewed directly on the console in text mode:

```
> maqao oneview --create-report=one -xp=maqao_npb \
--output-format=text
```

Optimising a code with MAQAO

Emmanuel OSERET

Matrix Multiply code

- "Naïve" dense matrix multiply implementation in C

```
void kernel0 (int n,
              float a[n][n],
              float b[n][n],
              float c[n][n]) {
  int i, j, k;

  for (i=0; i<n; i++)
    for (j=0; j<n; j++) {
      c[i][j] = 0.0f;
      for (k=0; k<n; k++)
        c[i][j] += a[i][k] * b[k][j];
    }
}
```

Preparing interactive session

- Request interactive session

```
> qsub -I -W group_list=PRACET1FWB_MAQAO -l walltime=0:30:00
```

- Load MAQAO environment

```
> module load parallel_tools/maqao/2.8.6
```

- Load latest GCC compiler

```
> module load compiler/gcc/7.2.0
```

- Go to matmul directory and compile matrix multiply codes

```
> cd $HOME/MAQAO_HANDSON/matmul
> make
```

Analysing matrix multiply with MAQAO

- Run baseline version of matrix multiply

```
> ./matmul_orig 150 10000 #usage: <nb repets> <matrix size>
cycles per FMA: 2.71
```

- Analyse matrix multiply with ONE View

```
> maqao oneview create-report=one \
binary=./matmul_orig run-command="<binary> 150 10000" \
xp=ov_orig
```

- OR, using a configuration script:

```
> maqao oneview create-report=one c=ov_orig.lua xp=ov_orig
```

Viewing results (HTML)

```
> tar -zcf $HOME/ov_orig.tgz ov_orig/RESULTS/matmul_orig_one_html
```

```
> scp -o ProxyCommand='ssh <username>@hpc.cenaero.be \
nc zenobe 22' <username>@zenobe:ov_orig.tgz .
> tar xf ov_orig.tgz
> firefox ov_orig/RESULTS/matmul_orig_one_html/index.html &
```

Global Metrics

Total Time (s)		36.8
Time in innermost loops (%)		99.9
Compilation Options		binary: -funroll-loops is missing.
Flow Complexity		1.00
Array Access Efficiency (%)		66.53
Clean	Potential Speedup	1.00
	Nb Loops to get 80%	1
FP Vectorised	Potential Speedup	2.11
	Nb Loops to get 80%	1
Fully Vectorised	Potential Speedup	7.94
	Nb Loops to get 80%	1

Viewing results (text)

```
> maqao oneview create-report=one xp=ov_orig \
  output-format=text --text-global | less
```

```
+-------------------------------------------------------------------------------+
+                                  1.2  -  Global Metrics                       +
+-------------------------------------------------------------------------------+

  Total Time:                          36.8 s
  Time spent in loops:                 99.49 %
  Compilation Options:                 binary:  -funroll-loops is missing.
  Flow Complexity:            1.00
  Array Access Efficiency:    66.67 %
  If Clean:
       Potential Speedup:     1.00
       Nb Loops to get 80%:   1
  If FP Vectorized:
       Potential Speedup:     2.11
       Nb Loops to get 80%:   1
  If Fully Vectorized:
       Potential Speedup:     7.94
       Nb Loops to get 80%:   1
```

Viewing results (text)

```
+-------------------------------------------------------------------------------+
+                                1.3  -  Potential Speedups                     +
+-------------------------------------------------------------------------------+

  If Clean:
      Number of loops    | 1       | 2       |
      Cumulated Speedup  | 1.0041  | 1.0041  |
  Top 5 loops:
    matmul_orig - 2:     1.0041
    matmul_orig - 1:     1.0041

  If FP Vectorized:
      Number of loops    | 1       | 2       |
      Cumulated Speedup  | 2.1113  | 2.1113  |
  Top 5 loops:
    matmul_orig - 1:     2.1113
    matmul_orig - 2:     2.1113

  If Fully Vectorized:
      Number of loops    | 1       | 2       |
      Cumulated Speedup  | 7.5160  | 7.9372  |
  Top 5 loops:
    matmul_orig - 1:     7.516
    matmul_orig - 2:     7.9372
```

Loop ID

Viewing CQA output (text)

```
> maqao oneview --create-report=one -xp=ov_orig \
  --output-format=text --text-cqa=1 | less
```

Loop ID

```
      Vectorization
  --------------------
Your loop is not vectorized.
8 data elements could be processed at once in vector registers.
By vectorizing your loop, you can lower the cost of an iteration from 3.00 to 0.38 cycles
(8.00x speedup).

Details
All SSE/AVX instructions are used in scalar version (process only one data element in
vector registers).
Since your execution units are vector units, only a vectorized loop can use their full
power.

Workaround
 - Try another compiler or update/tune your current one:
  * recompile with fassociative-math (included in Ofast or ffast-math) to extend loop
vectorization to FP reductions.
 - (…)
```

CQA output for the baseline kernel

Vectorization

Your loop is not vectorized. 8 data elements could be processed at once in vector registers. By vectorizing your loop, you can lower the cost of an iteration from 3.00 to 0.38 cycles (8.00x speedup).

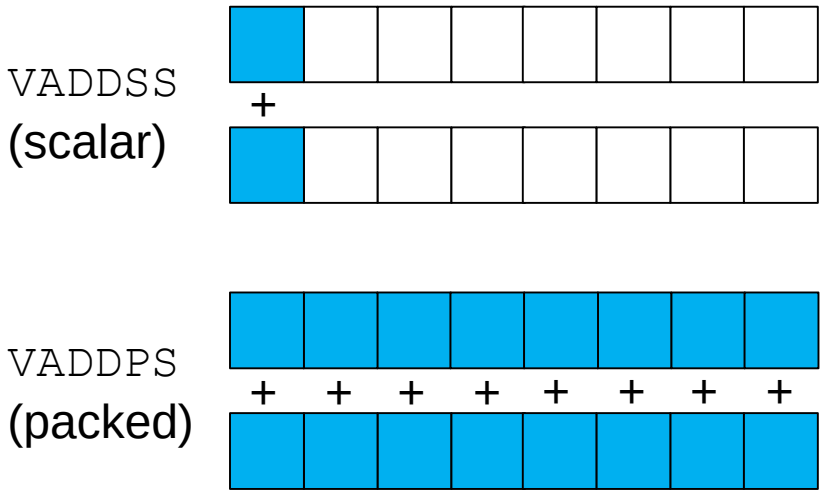
Details

All SSE/AVX instructions are used in scalar version (process only one data element in vector registers). Since your execution units are vector units, only a vectorized loop can use their full power.

Workaround

- Try another compiler or update/tune your current one:
  - recompile with fassociative-math (included in Ofast or ffast-math) to extend loop vectorization to FP reductions.
- Remove inter-iterations dependences from your loop and make it unit-stride:
  - If your arrays have 2 or more dimensions, check whether elements are accessed contiguously and, otherwise, try to permute loops accordingly: C storage order is row-major: for(i) for(j) a[j][i] = b[j][i]; (slow, non stride 1) => for(i) for(j) a[i][j] = b[i][j]; (fast, stride 1)
  - If your loop streams arrays of structures (AoS), try to use structures of arrays instead (SoA): for(i) a[i].x = b[i].x; (slow, non stride 1) => for(i) a.x[i] = b.x[i]; (fast, stride 1)

Vectorization (summing elements):

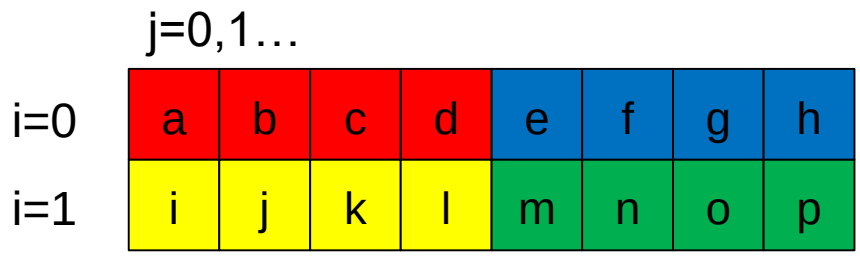


Accesses are not contiguous => let's permute k and j loops

No structures here…

Impact of loop permutation on data access

Logical mapping

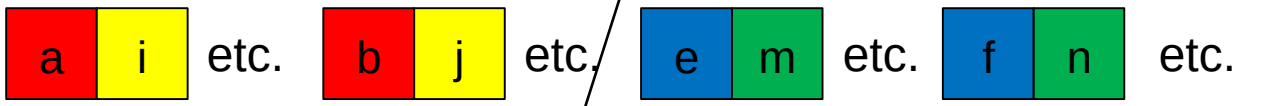


Efficient vectorization + prefetching

Physical mapping

(C stor. order: row-major)


```
for (j=0; j<n; j++)
  for (i=0; i<n; i++)
    f(a[i][j]);
```




```
for (i=0; i<n; i++)
  for (j=0; j<n; j++)
    f(a[i][j]);
```

Removing inter-iteration dependences and getting stride 1 by permuting loops on j and k

```
void kernel1 (int n,
              float a[n][n],
              float b[n][n],
              float c[n][n]) {
  int i, j, k;

  for (i=0; i<n; i++) {
    for (j=0; j<n; j++)
      c[i][j] = 0.0f;

    for (k=0; k<n; k++)
      for (j=0; j<n; j++)
        c[i][j] += a[i][k] * b[k][j];
  }
}
```

Analyse matrix multiply with permuted loops

- Run permuted loops version of matrix multiply

```
> ./matmul_perm 150 10000
cycles per FMA: 0.61
```

- Analyse matrix multiply with ONE View

```
> maqao oneview create-report=one \
binary=./matmul_perm run-command="<binary> 150 10000" \
xp=ov_perm
```

- OR using configuration script:

```
> maqao oneview create-report=one c=ov_perm.lua xp=ov_perm
```

- Viewing new results

```
> maqao oneview create-report=one xp=ov_perm \
  output-format=text --text-global | less
```

(Or download the ov_perm/RESULTS/matmul_perm_one_html folder locally and open ov_perm/RESULTS/matmul_perm_one_html/index.html)

Loop permutation results

Faster (was 36.8)

Global Metrics

Total Time (s)		8.24
Time in innermost loops (%)		99.51
Compilation Options		binary: -funroll-loops is missing.
Flow Complexity		1.00
Array Access Efficiency (%)		75.00
Clean	Potential Speedup	1.22
	Nb Loops to get 80%	2
FP Vectorised	Potential Speedup	1.36
	Nb Loops to get 80%	2
Fully Vectorised	Potential Speedup	2.20
	Nb Loops to get 80%	2

More efficient vectorization (was 7.94)

Loop permutation results

Faster (was 36.8)

Let's try this

Global Metrics

Total Time (s)		8.24
Time in innermost loops (%)		99.51
Compilation Options		binary: -funroll-loops is missing.
Flow Complexity		1.00
Array Access Efficiency (%)		75.00
Clean	Potential Speedup	1.22
	Nb Loops to get 80%	2
FP Vectorised	Potential Speedup	1.36
	Nb Loops to get 80%	2
Fully Vectorised	Potential Speedup	2.20
	Nb Loops to get 80%	2

More efficient vectorization (was 7.94)

CQA output after loop permutation

```
> maqao oneview create-report=one xp=ov_perm \
  output-format=text text-cqa=4 | less
```

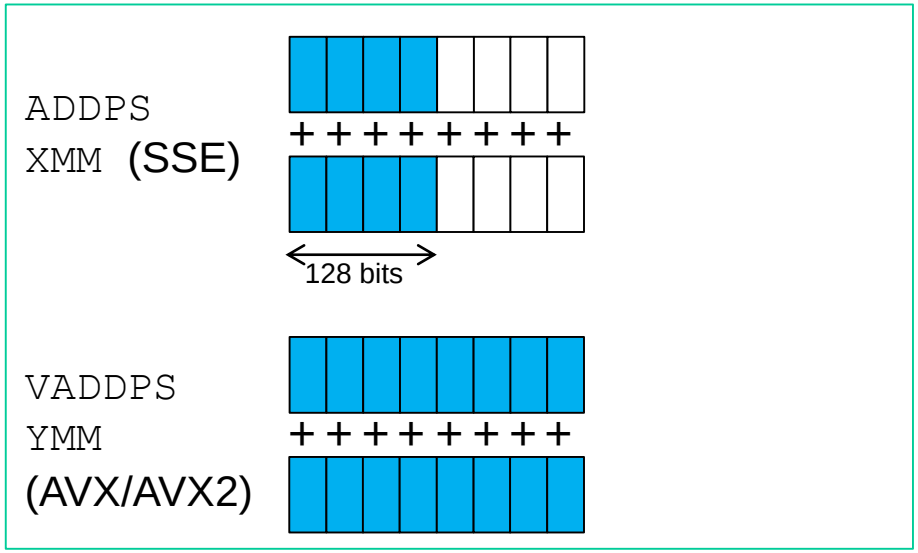
```
      Vectorization
  --------------------
Your loop is vectorized, but using only 128 out of 256 bits (SSE/AVX-128 instructions on
AVX/AVX2 processors).
By fully vectorizing your loop, you can lower the cost of an iteration from 1.75 to 0.88
cycles (4.00x speedup).

Details
All SSE/AVX instructions are used in vector version (process two or more data elements in
vector registers).
Since your execution units are vector units, only a fully vectorized loop can use their
full power.

Workaround
 - Recompile with march=core-avx2.
CQA target is Core_i7X_Xeon_E5_v3_E7_v3 (Intel Xeon processor E5-4600/2600/1600 v3 product
families…) but specialization flags are -march=x86-64
 - (…)
```

Let's add –

march=core-avx2

Impacts of architecture specialization: vectorization and FMA

- Vectorization
  - SSE instructions (SIMD 128 bits) used on a processor supporting AVX/AVX2 ones (SIMD 256 bits)
  - => 50% efficiency loss



- FMA
  - Fused Multiply-Add (A+BC)
  - Intel architectures: supported on MIC/KNC and Xeon starting from Haswell

```
# A = A + BC

VMULPS <B>,<C>,%XMM0
VADDPS <A>,%XMM0,<A>
# can be replaced with
something like:
VFMADD312PS <B>,<C>,<A>
```

Analyse matrix multiply with architecture specialisation

- Run architecture specialisation version of matrix multiply

```
> ./matmul_perm_opt 150 10000
cycles per FMA: 0.44
```

- Analyse matrix multiply with ONE View

```
> maqao oneview create-report=one \
binary=./matmul_perm_opt run-command="<binary> 150 10000" \
xp=ov_perm_opt
```

- OR using configuration script:

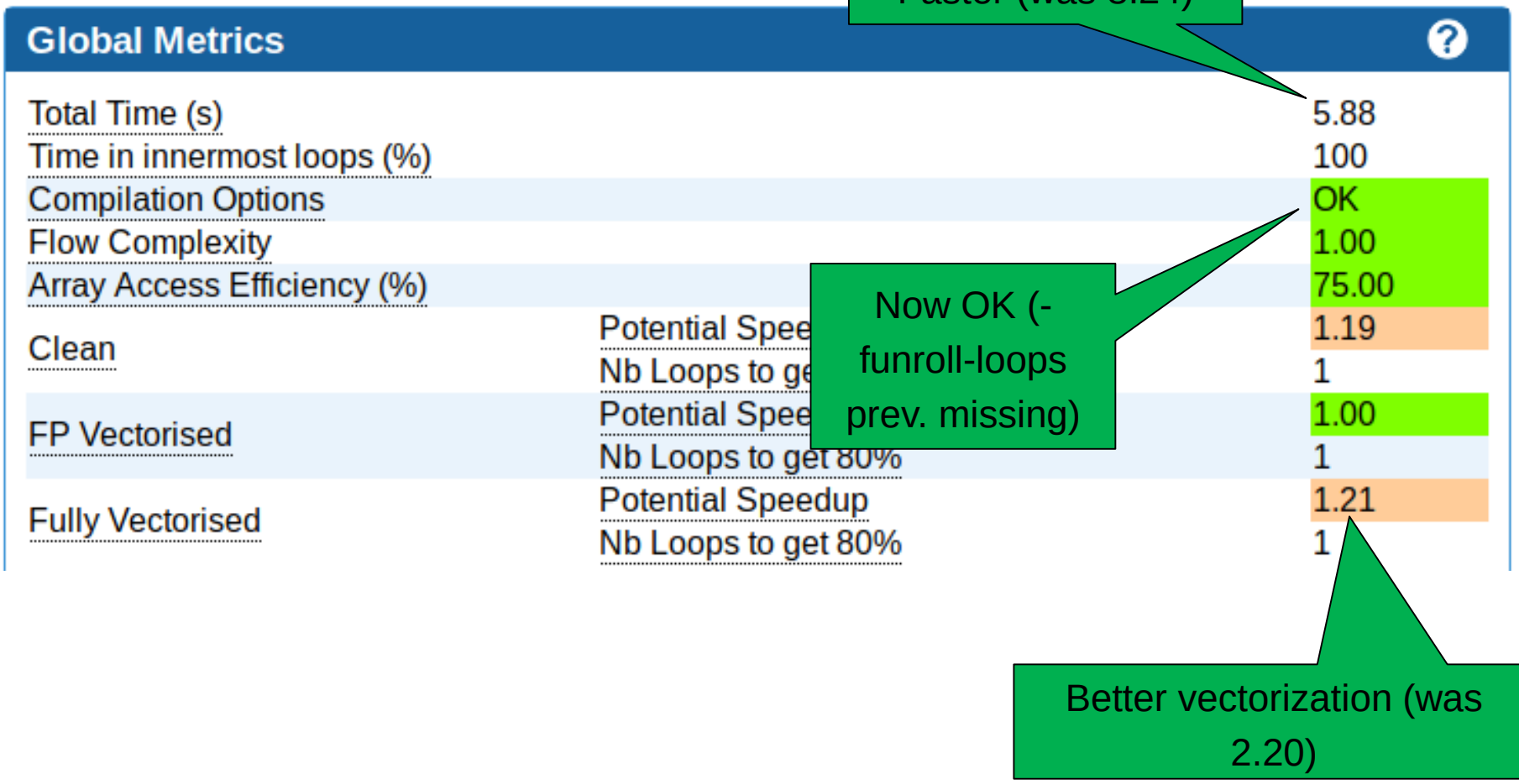
```
> maqao oneview create-report=one c=ov_perm_opt.lua xp=ov_perm_opt
```

- Viewing new results:

```
> maqao oneview create-report=one xp=ov_perm_opt \
  output-format=text --text-global | less
```

- (or download the ov_perm_opt/RESULTS/matmul_perm_opt_one_html folder locally and open index.html in your browser)

Loop permutation + (-march=core-avx2 -funroll-loops)



CQA output with (-march=core-avx2 -funroll-loops)

Vectorization

Your loop is fully vectorized, using full register length.

Details

All SSE/AVX instructions are used in vector version (process two or more data elements in vector registers).

Workaround

Use vector aligned instructions:

1. align your arrays on 32 bytes boundaries: replace { void *p = malloc (size); } with { void *p; posix_memalign (&p, 32, size); }.
2. inform your compiler that your arrays are vector aligned: if array 'foo' is 32 bytes-aligned, define a pointer 'p_foo' as __builtin_assume_aligned (foo, 32) and use it instead of 'foo' in the loop.

Let's switch to the next proposal: vector aligned instructions

```
> maqao oneview create-report=one xp=ov_perm_opt \
  output-format=text text-cqa=4 | less
```

Using aligned arrays in matrix multiply

- Run aligned version of matrix multiply

```
> ./matmul_align 150 10000
Cannot call kernel on matrices with size%8 != 0 (data not
aligned on 32B boundaries)
Aborted
```

- => Alignment imposes restrictions on input parameters.

```
> ./matmul_align 152 10000
driver.c: Using posix_memalign instead of malloc
cycles per FMA: 0.24
```

Analysing matrix multiply with aligned arrays

- Analyse matrix multiply with ONE View

```
> maqao oneview create-report=one \
binary=./matmul_align run-command="<binary> 152 10000" \
xp=ov_align
```

- OR using configuration script:

```
> maqao oneview create-report=one c=ov_align.lua \
 xp=ov_align
```

- Viewing new results

```
> maqao oneview create-report=one xp=ov_align \
  output-format=text --text-global | less
```

(Or download the ov_align/RESULTS/matmul_align_one_html folder locally and open ov_align/RESULTS/matmul_align_one_html/index.html in your browser)

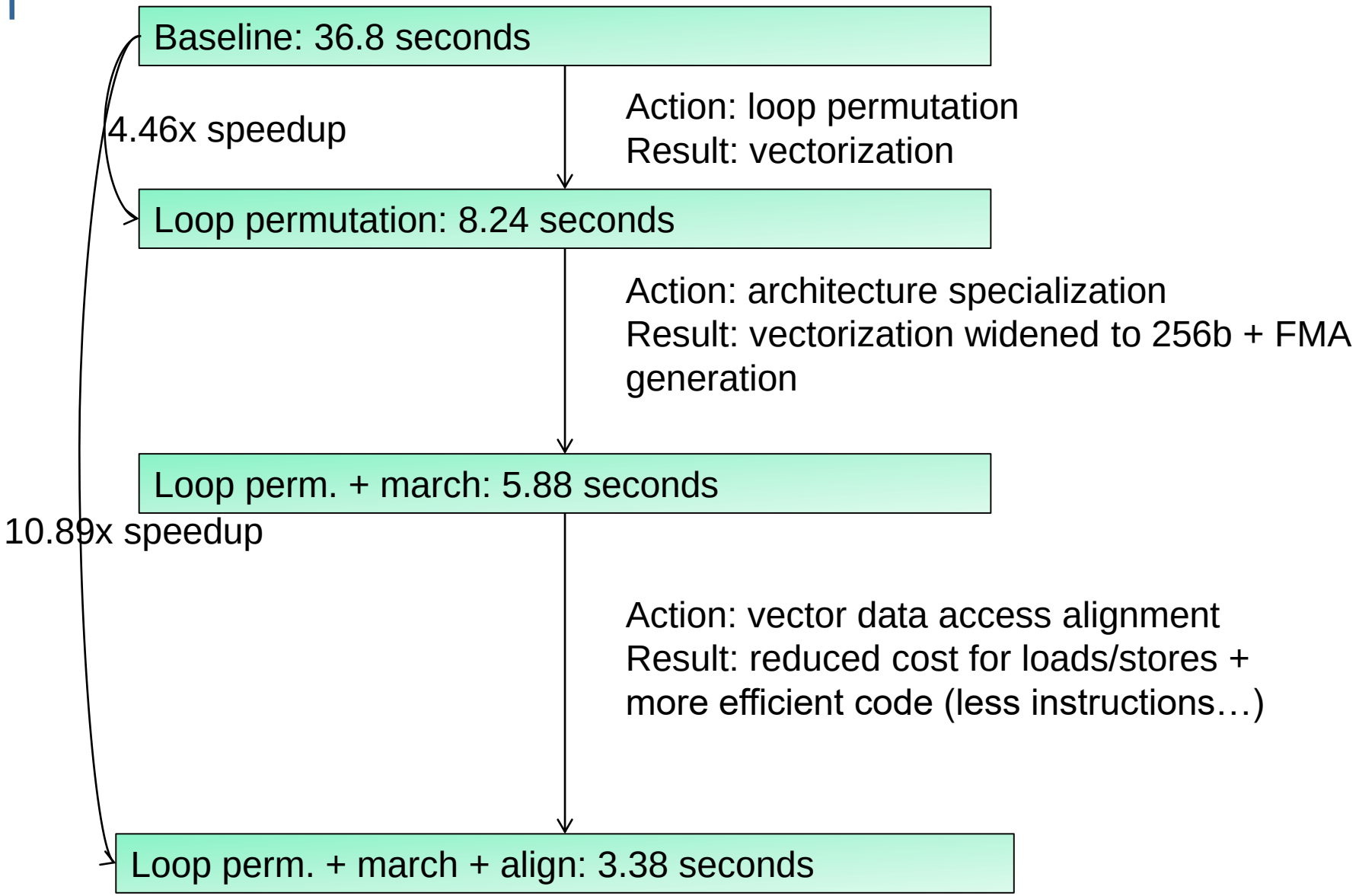
Vector-aligning array accesses

Global Metrics

Total Time (s)		3.38
Time in innermost loops (%)		98.81
Compilation Options		OK
Flow Complexity		1.00
Array Access Efficiency (%)		75.00
Clean	Potential Speedup	1.15
	Nb Loops to get 80%	1
FP Vectorised	Potential Speedup	1.00
	Nb Loops to get 80%	1
Fully Vectorised	Potential Speedup	1.13
	Nb Loops to get 80%	1

Extra speedup (was 5.88)

Summary of optimizations and gains



- (If necessary) Request interactive session and load MAQAO environment

```
> qsub -I -W group_list=PRACET1FWB_MAQAO -l walltime=0:30:00
> module load parallel_tools/maqao/2.8.6
```

- Switch to the hydro handson folder

```
> cd $HOME/MAQAO_HANDSON/hydro
```

- Load Intel compiler environment

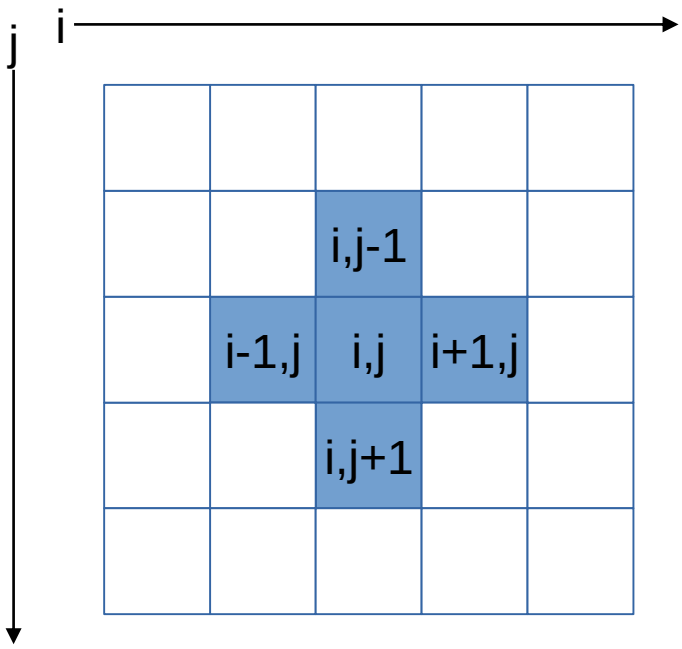
```
> module load compiler/intel/2018.0.1
```

- Compile

```
> make
```

```
int build_index (int i, int j, int grid_size)
{
  return (i + (grid_size + 2) * j);
}

void linearSolver0 (...) {
  int i, j, k;

  for (k=0; k<20; k++)
    for (i=1; i<=grid_size; i++)
      for (j=1; j<=grid_size; j++)
        x[build_index(i, j, grid_size)] =
  (a * ( x[build_index(i-1, j, grid_size)] +
         x[build_index(i+1, j, grid_size)] +
         x[build_index(i, j-1, grid_size)] +
         x[build_index(i, j+1, grid_size)]
       ) + x0[build_index(i, j, grid_size)]
  ) / c;
}
```

Iterative linear system solver using the Gauss-Siedel relaxation technique.
« Stencil » code



Running and analyzing kernel0 (icc -O3 -xHost)

- Run baseline kernel

```
> ./hydro_k0 300 200
Cycles per element for solvers: 2493.81
```

- Profile with MAQAO

```
> maqao oneview create-report=one xp=ov_k0 c=ov_k0.lua
```

- Display results

```
> maqao oneview create-report=one xp=ov_k0 \
output-format=text --text-global | less
```

```
+-------------------------------------------------------------+
+                      1.2  -  Global Metrics                  +
+-------------------------------------------------------------+

  Total Time:                          17.18 s
  Time spent in loops:                 99.46 %
  Compilation Options:                 OK
  Flow Complexity:                     1.09
  Array Access Efficiency:             25.24 %
```

The kernel routine, linearSolver, was inlined in caller functions.
Moreover, there is direct mapping between source and binary loop.
Consequently the 4 hot loops are identical and only one needs analysis.

Loops Index

Coverage (%) Level Time (s) Vectorization Ratio (%) Speedup If Clean
Speedup If FP Vectorized Speedup If Fully Vectorized Select all

Loop id	Source Location	Source Function	Coverage (%)	Level
132	hydro_k0:kernel.c:104-110	project	29.19	Innermost
85	hydro_k0:kernel.c:104-110	c_velocitySolver	20.26	Innermost
92	hydro_k0:kernel.c:104-110	c_velocitySolver	20.26	Innermost
53	hydro_k0:kernel.c:104-110	c_densitySolver	19.61	Innermost
134	hydro_k0:kernel.c:380-383	project	2.61	Innermost
44	hydro_k0:kernel.c:15-292	c_densitySolver	1.74	Innermost
73	hydro_k0:kernel.c:15-292	c_velocitySolver	1.31	Innermost
121	hydro_k0:kernel.c:210-342	c_velocitySolver	1.31	Innermost
136	hydro_k0:kernel.c:368-371	project	1.31	Innermost
71	hydro_k0:kernel.c:15-292	c_velocitySolver	1.09	Innermost
107	hydro_k0:kernel.c:210-318	c_velocitySolver	0.44	Innermost
106	hydro_k0:kernel.c:239-241	c_velocitySolver	0.44	Innermost
78	hydro_k0:kernel.c:28-32	c_velocitySolver	0.22	Single

Loop Id: 129 | **Module: hydro_k0** | **Source: kernel.c:104-110** | **Coverage: 31.23%**

Source Code

```
/gpfs/home/nct00/nct00010/TESTS_HANDSON/MAQAO_HANDSON/hydro//kernel.c: 104 - 110
------------------------------------------------------------------------------

104:       for (j = 1; j <= grid_size; j++)
105:         {
106:         x[build_index(i, j, grid_size)] = (a * ( x[build_index(i-1, j, grid_size)] +
107:                                                  x[build_index(i+1, j, grid_size)] +
108:                                                  x[build_index(i, j-1, grid_size)] +
109:                                                  x[build_index(i, j+1, grid_size)]) +
110:                                        x0[build_index(i, j, grid_size)]) / c;
111:         }
112:       }
```

CQA

The related source loop is not unrolled or unrolled with no peel/tail loop.

gain | potential | hint | expert

Vectorization

Your loop is not vectorized. Only 6% of vector register length is used (average across all SSE/AVX instructions). By vectorizing your loop, you can lower the cost of an iteration from 4.00 to 0.25 cycles (16.00x speedup).

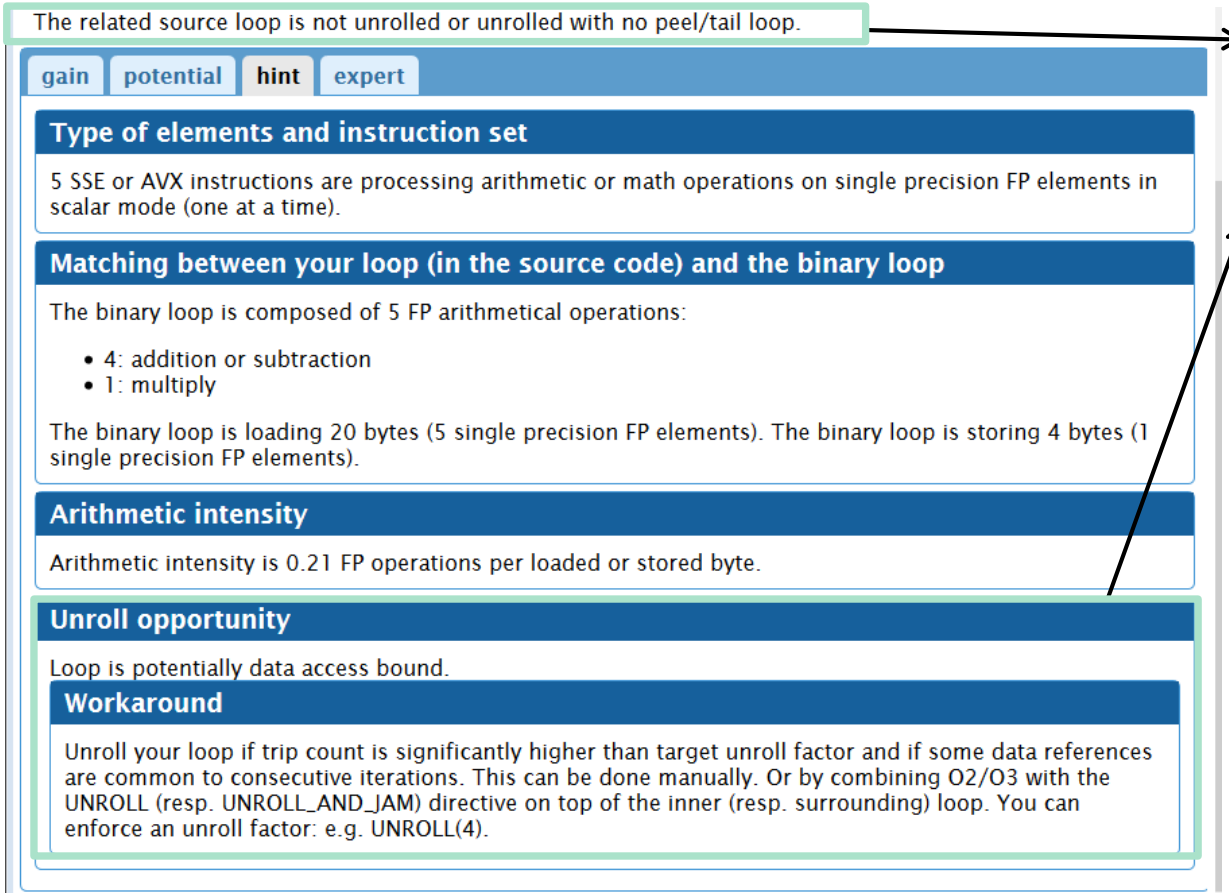
Details

All SSE/AVX instructions are used in scalar version (process only one data element in vector registers). Since your execution units are vector units, only a vectorized loop can use their full power.

Workaround

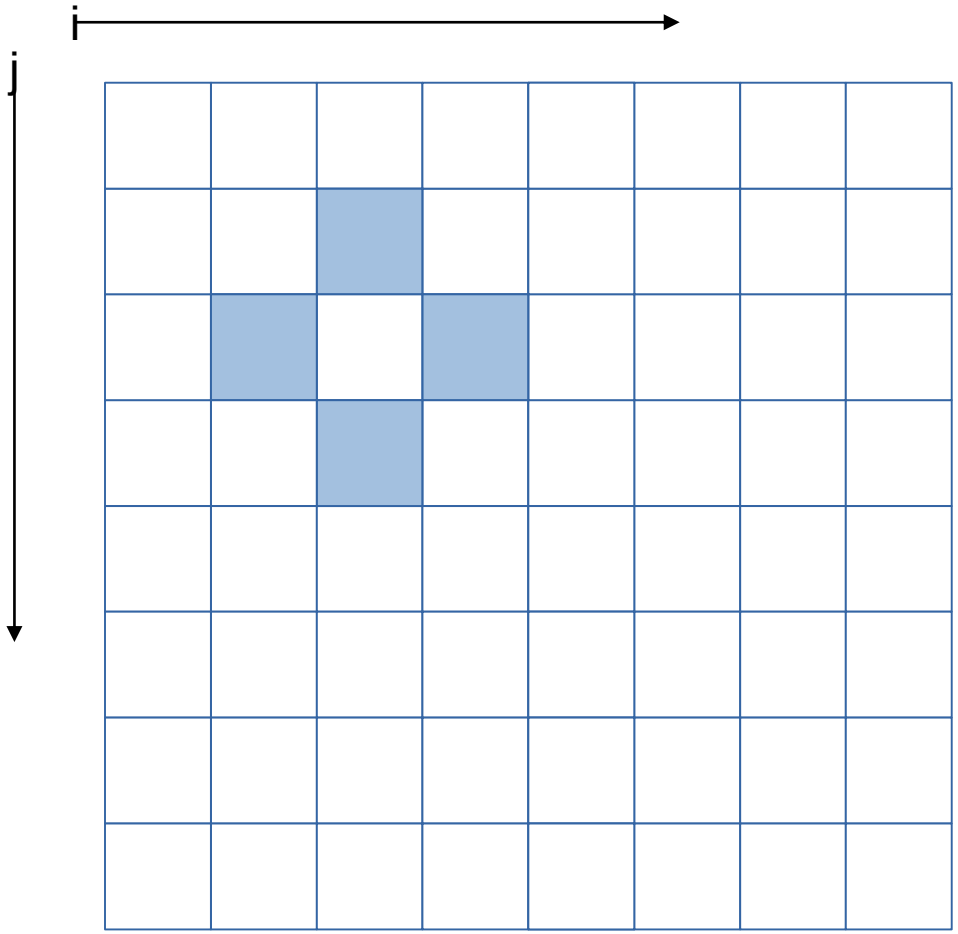
- Try another compiler or update/tune your current one:
  - use the vec-report option to understand why your loop was not vectorized. If "existence of vector dependences", try the IVDEP directive. If, using IVDEP, "vectorization possible but seems inefficient", try the VECTOR ALWAYS directive.
- Remove inter-iterations dependences from your loop and make it unit-stride:
  - If your arrays have 2 or more dimensions, check whether elements are accessed contiguously and, otherwise, try to permute loops accordingly: C storage order is row-major: for(i) for(j) a[j][i] = b[j][i]; (slow, non stride 1) => for(i) for(j) a[i][j] = b[i][j]; (fast, stride 1)
  - If your loop streams arrays of structures (AoS), try to use structures of arrays instead

CQA output for kernel0



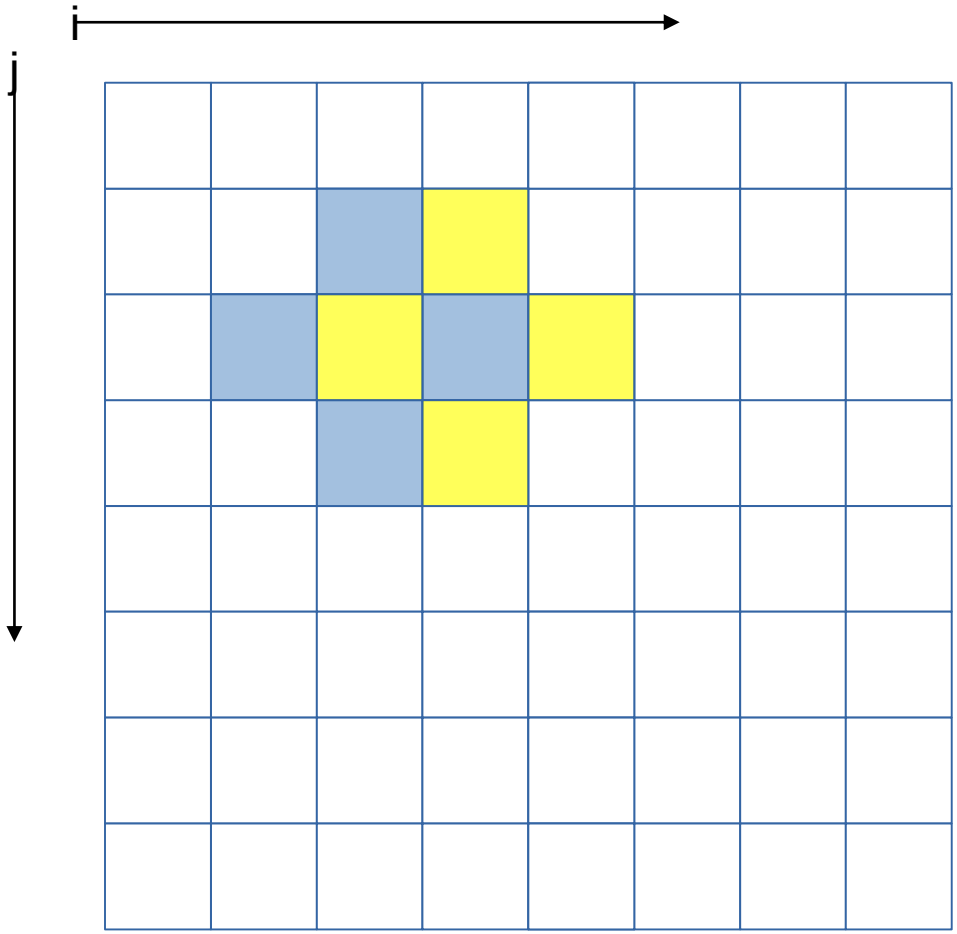
Unrolling is generally a good deal: fast to apply and often provides gain. Let's try to reuse data references through unrolling

Memory references reuse : 4x4 unroll footprint on loads



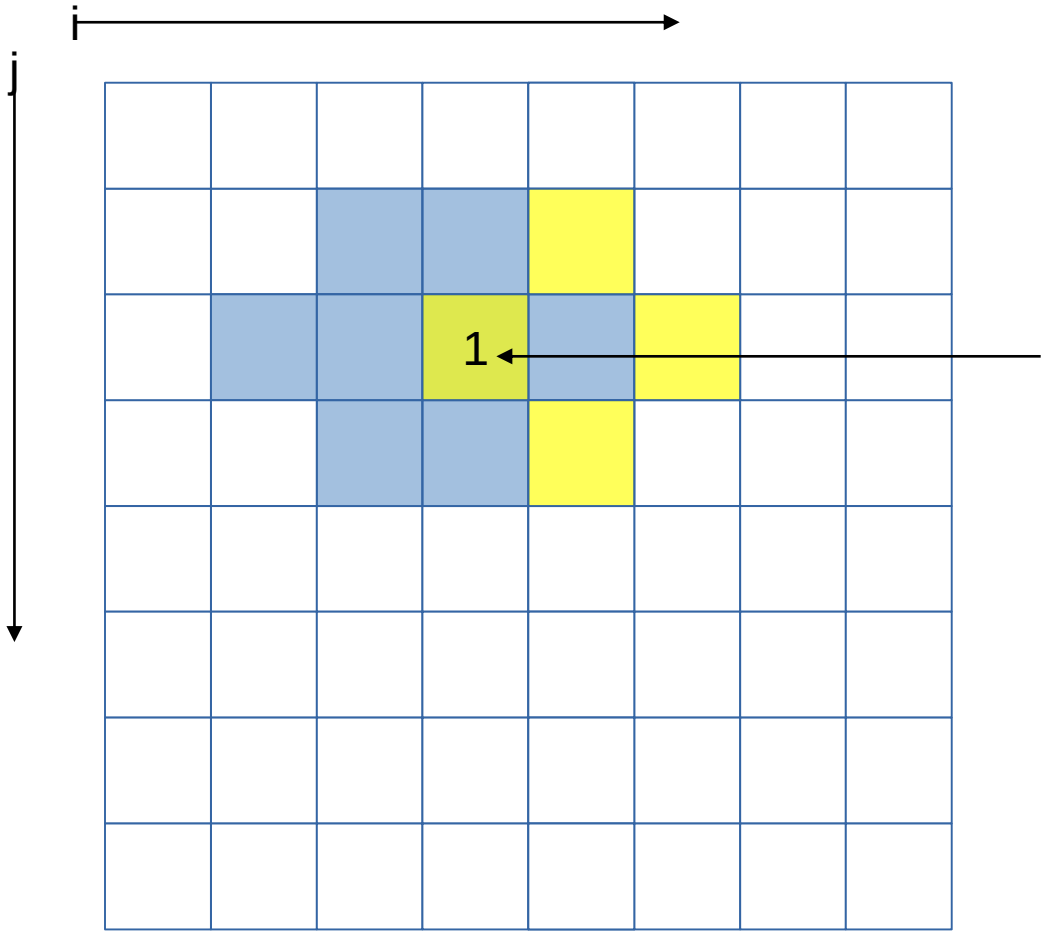

```
LINEAR_SOLVER(i+0,j+0)
```

Memory references reuse : 4x4 unroll footprint on loads




```
LINEAR_SOLVER(i+0,j+0)
LINEAR_SOLVER(i+1,j+0)
```

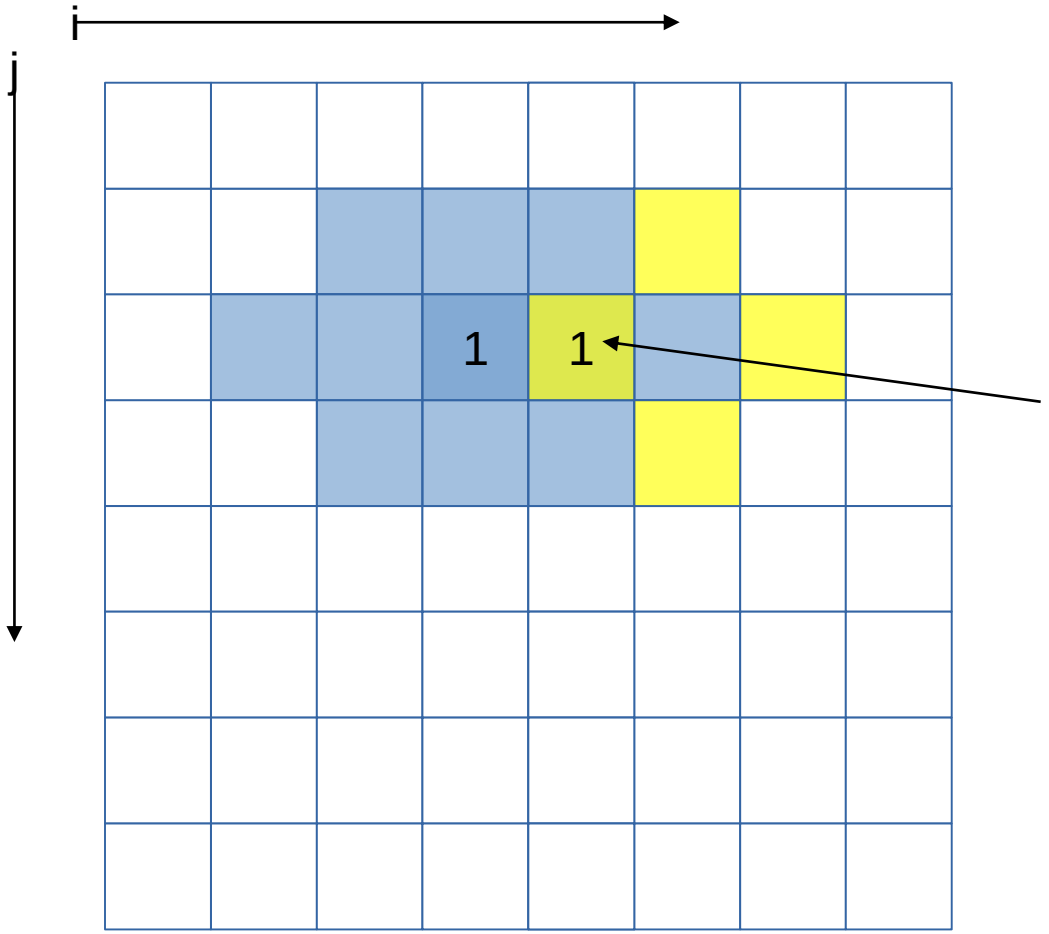
Memory references reuse : 4x4 unroll footprint on loads




```
LINEAR_SOLVER(i+0,j+0)
LINEAR_SOLVER(i+1,j+0)
LINEAR_SOLVER(i+2,j+0)
```

1 reuse

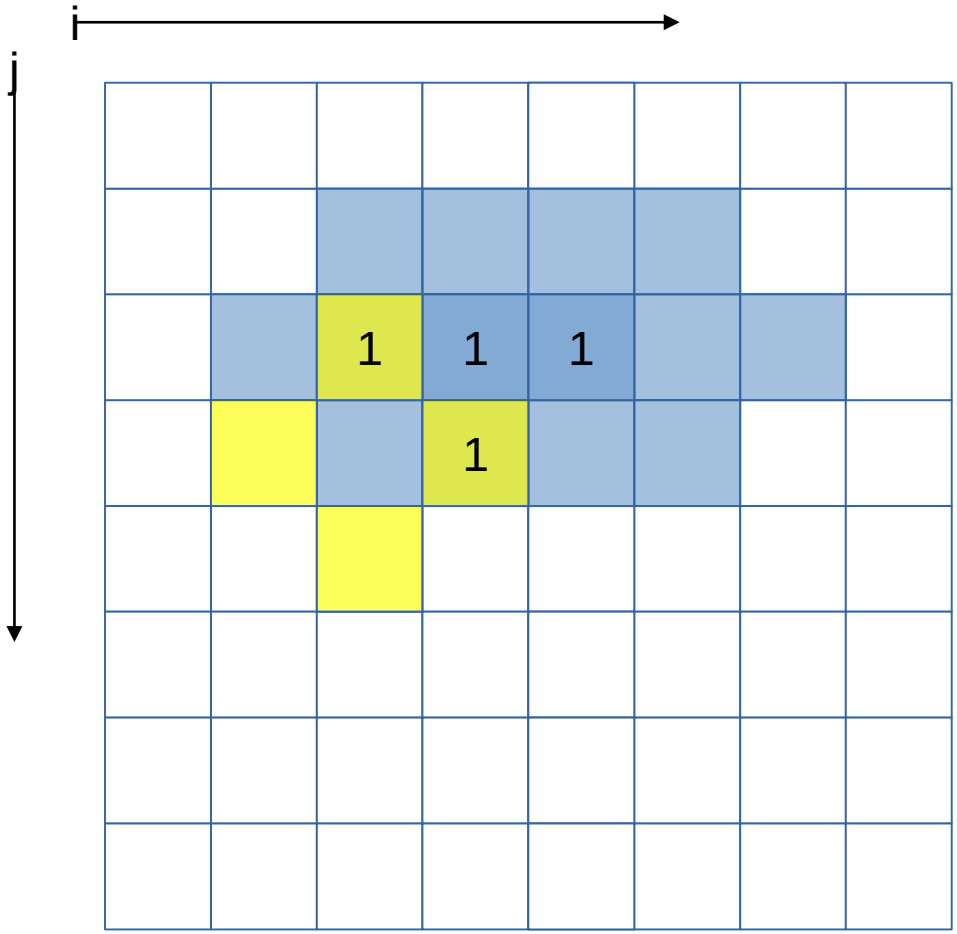
Memory references reuse : 4x4 unroll footprint on loads




```
LINEAR_SOLVER(i+0,j+0)
LINEAR_SOLVER(i+1,j+0)
LINEAR_SOLVER(i+2,j+0)
LINEAR_SOLVER(i+3,j+0)
```

2 reuses

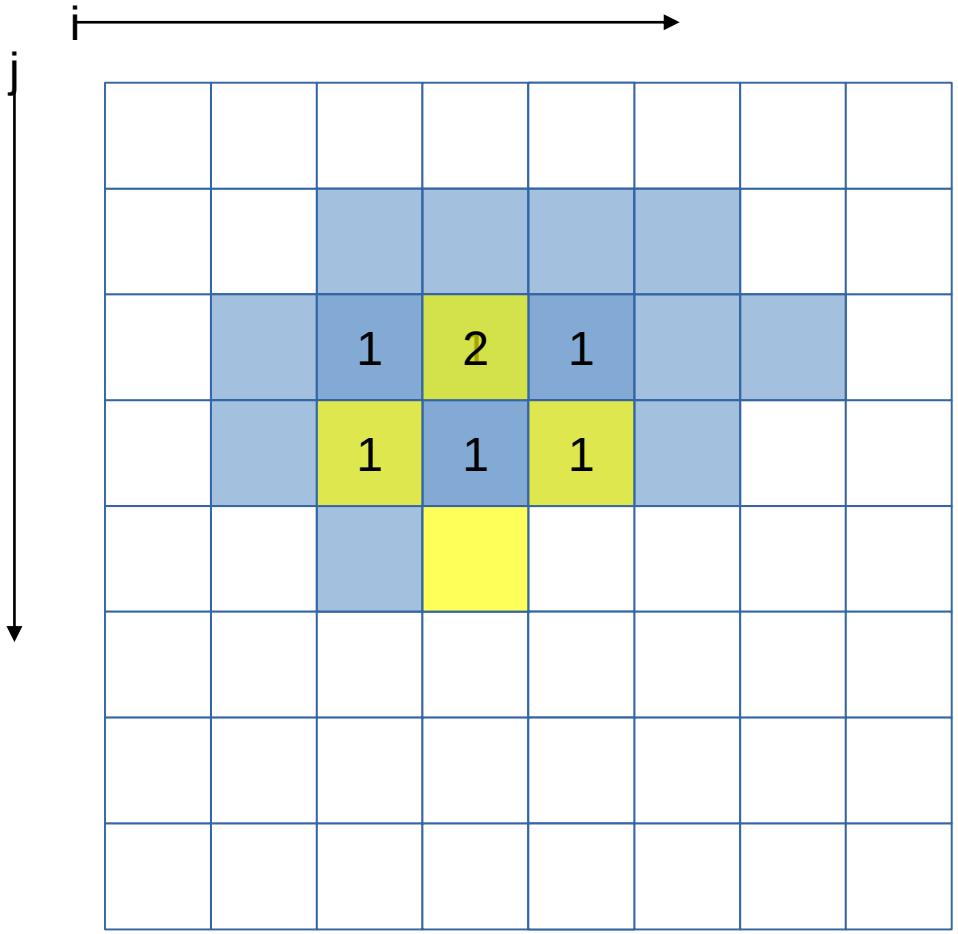
Memory references reuse : 4x4 unroll footprint on loads




```
LINEAR_SOLVER(i+0,j+0)
LINEAR_SOLVER(i+1,j+0)
LINEAR_SOLVER(i+2,j+0)
LINEAR_SOLVER(i+3,j+0)

LINEAR_SOLVER(i+0,j+1)
```

4 reuses

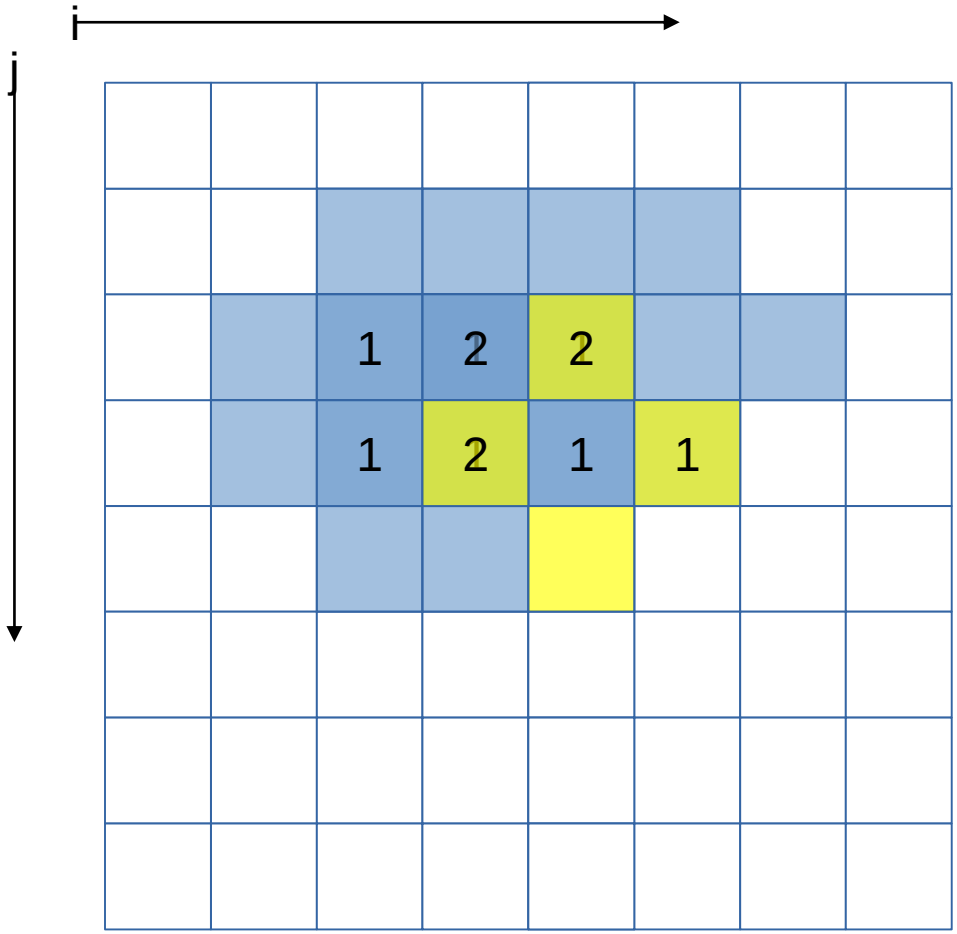
Memory references reuse : 4x4 unroll footprint on loads




```
LINEAR_SOLVER(i+0,j+0)
LINEAR_SOLVER(i+1,j+0)
LINEAR_SOLVER(i+2,j+0)
LINEAR_SOLVER(i+3,j+0)

LINEAR_SOLVER(i+0,j+1)
LINEAR_SOLVER(i+1,j+1)
```

7 reuses

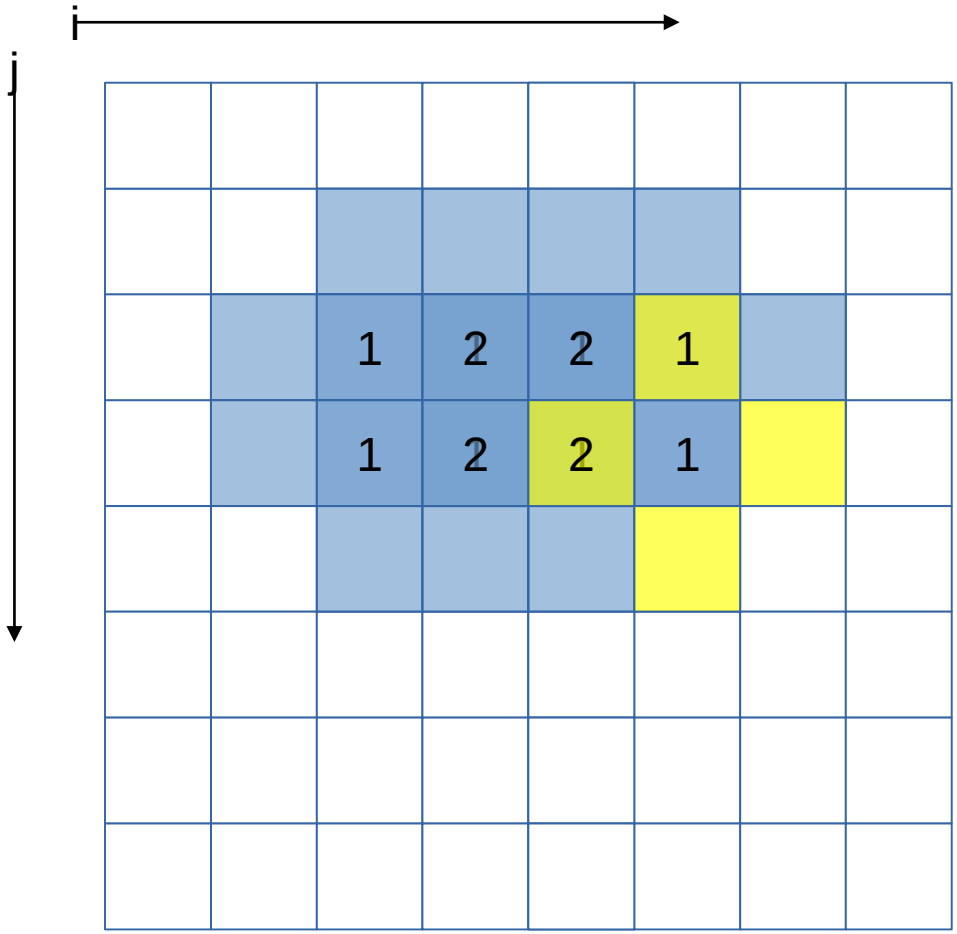
Memory references reuse : 4x4 unroll footprint on loads




```
LINEAR_SOLVER(i+0,j+0)
LINEAR_SOLVER(i+1,j+0)
LINEAR_SOLVER(i+2,j+0)
LINEAR_SOLVER(i+3,j+0)

LINEAR_SOLVER(i+0,j+1)
LINEAR_SOLVER(i+1,j+1)
LINEAR_SOLVER(i+2,j+1)
```

10 reuses

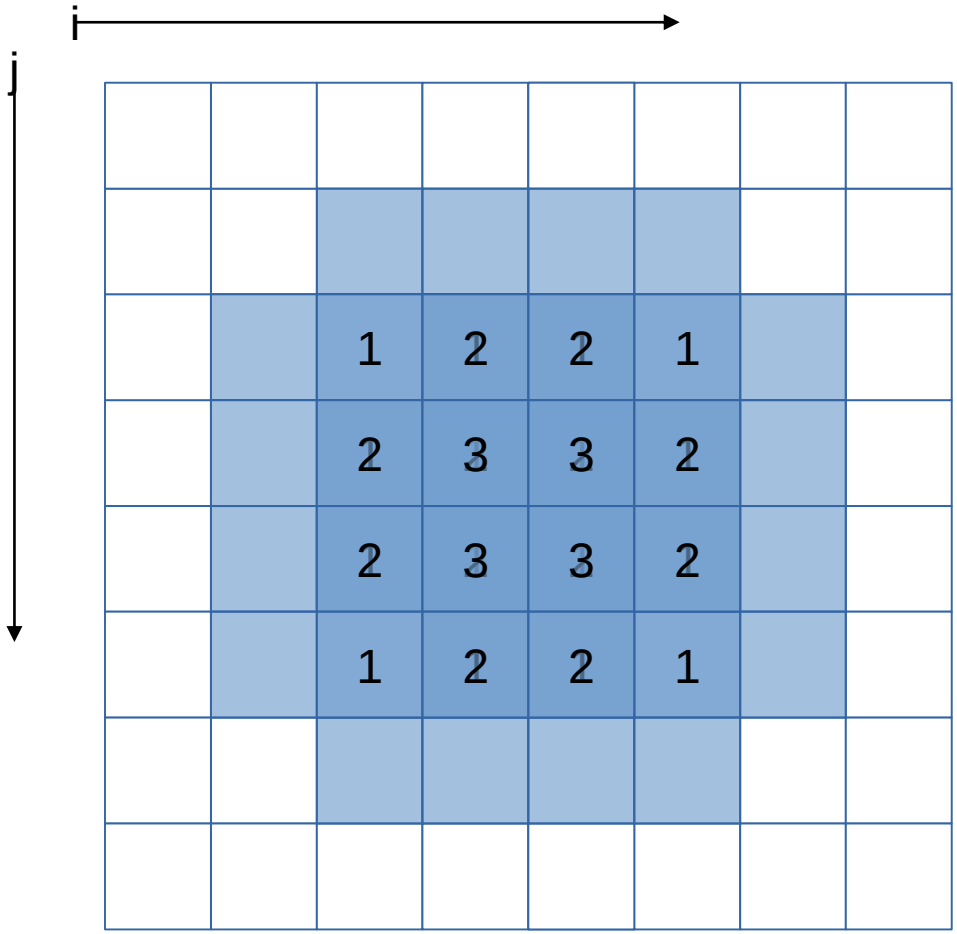
Memory references reuse : 4x4 unroll footprint on loads




```
LINEAR_SOLVER(i+0,j+0)
LINEAR_SOLVER(i+1,j+0)
LINEAR_SOLVER(i+2,j+0)
LINEAR_SOLVER(i+3,j+0)

LINEAR_SOLVER(i+0,j+1)
LINEAR_SOLVER(i+1,j+1)
LINEAR_SOLVER(i+2,j+1)
LINEAR_SOLVER(i+3,j+1)
```

12 reuses

Memory references reuse : 4x4 unroll footprint on loads




```
LINEAR_SOLVER(i+0-3,j+0)
LINEAR_SOLVER(i+0-3,j+1)
```

`LINEAR_SOLVER(i+0-3,j+2)`

`LINEAR_SOLVER(i+0-3,j+3)`

32 reuses

Impacts of memory reuse

- For the x array, instead of 4x4x4 = 64 loads, now only 32 (32 loads avoided by reuse)
- For the x0 array no reuse possible : 16 loads
- Total loads : 48 instead of 80

4x4 unroll

```
#define LINEARSOLVER(...) x[build_index(i, j, grid_size)] = …

void linearSolver2 (...) {
  (...)

  for (k=0; k<20; k++)
    for (i=1; i<=grid_size-3; i+=4)
      for (j=1; j<=grid_size-3; j+=4) {
        LINEARSOLVER (…, i+0, j+0);
        LINEARSOLVER (…, i+0, j+1);
        LINEARSOLVER (…, i+0, j+2);
        LINEARSOLVER (…, i+0, j+3);

        LINEARSOLVER (…, i+1, j+0);
        LINEARSOLVER (…, i+1, j+1);
        LINEARSOLVER (…, i+1, j+2);
        LINEARSOLVER (…, i+1, j+3);

        LINEARSOLVER (…, i+2, j+0);
        LINEARSOLVER (…, i+2, j+1);
        LINEARSOLVER (…, i+2, j+2);
        LINEARSOLVER (…, i+2, j+3);

        LINEARSOLVER (…, i+3, j+0);
        LINEARSOLVER (…, i+3, j+1);
        LINEARSOLVER (…, i+3, j+2);
        LINEARSOLVER (…, i+3, j+3);
      }
}
```

grid_size must now be multiple of 4. Or loop control must be adapted (much less readable) to handle leftover iterations

Running and analyzing kernel1

- Run optimized kernel

```
> ./hydro_k1 300 200
Cycles per element for solvers: 769.10
```

- Profile with MAQAO

```
> maqao oneview create-report=one xp=ov_k1 c=ov_k1.lua
```

- Display results

```
> maqao oneview create-report=one xp=ov_k1 \
output-format=text --text-global | less
```

```
+-------------------------------------------------------------+
+                      1.2  -  Global Metrics                  +
+-------------------------------------------------------------+

  Total Time:                          5.84 s
  Time spent in loops:                 97.97 %
  Compilation Options:                 OK
  Flow Complexity:                     1.29
  Array Access Efficiency:             26.23 %
```

Loops Index

Coverage (%) Level Time (s) Vectorization Ratio (%) Speedup If Clean
Speedup If FP Vectorized Speedup If Fully Vectorized Select all

Loop id	Source Location	Source Function	Coverage (%)	Level
138	hydro_k1:kernel.c:15-176	linearSolver1	58.56	Innermost
72	hydro_k1:kernel.c:15-292	c_velocitySolver	4.11	Innermost
45	hydro_k1:kernel.c:15-292	c_densitySolver	3.77	Innermost
124	hydro_k1:kernel.c:210-342	c_velocitySolver	3.42	Innermost
76	hydro_k1:kernel.c:368-371	c_velocitySolver	2.74	Innermost
70	hydro_k1:kernel.c:380-383	c_velocitySolver	2.4	Innermost
74	hydro_k1:kernel.c:15-292	c_velocitySolver	1.71	Innermost
110	hydro_k1:kernel.c:210-318	c_velocitySolver	1.71	Innermost
88	hydro_k1:kernel.c:380-383	c_velocitySolver	1.37	Innermost
90	hydro_k1:kernel.c:368-371	c_velocitySolver	1.37	Innermost
68	hydro_k1:kernel.c:456-459	c_velocitySolver	0.68	Single
100	hydro_k1:kernel.c:28-32	c_velocitySolver	0.68	Single

Remark: less calls were unrolled since linearSolver is now much more bigger

Loop Id: 135 **Module: hydro_k1** **Source: kernel.c:15-176** **Coverage: 63.14%**

Source Code

```
146:
147: void linearSolver1(int b, float* x, float* x0, float a, float c, float d
148:   {
149:   int i,j,k;
150:   const float inv_c = 1.0f / c;
151:
152:   for (k = 0; k < 20; k++)
153:     {
154:     for (i = 1; i <= grid_size-3; i+=4)
155:       {
156:       for (j = 1; j <= grid_size-3; j+=4)
157:     {
158:       LINEARSOLVER (x, x0, a, inv_c, grid_size, i+0, j+0);
159:       LINEARSOLVER (x, x0, a, inv_c, grid_size, i+0, j+1);
160:       LINEARSOLVER (x, x0, a, inv_c, grid_size, i+0, j+2);
161:       LINEARSOLVER (x, x0, a, inv_c, grid_size, i+0, j+3);
162:
163:       LINEARSOLVER (x, x0, a, inv_c, grid_size, i+1, j+0);
164:       LINEARSOLVER (x, x0, a, inv_c, grid_size, i+1, j+1);
165:       LINEARSOLVER (x, x0, a, inv_c, grid_size, i+1, j+2);
166:       LINEARSOLVER (x, x0, a, inv_c, grid_size, i+1, j+3);
167:
168:       LINEARSOLVER (x, x0, a, inv_c, grid_size, i+2, j+0);
169:       LINEARSOLVER (x, x0, a, inv_c, grid_size, i+2, j+1);
170:       LINEARSOLVER (x, x0, a, inv_c, grid_size, i+2, j+2);
171:       LINEARSOLVER (x, x0, a, inv_c, grid_size, i+2, j+3);
172:
173:       LINEARSOLVER (x, x0, a, inv_c, grid_size, i+3, j+0);
174:       LINEARSOLVER (x, x0, a, inv_c, grid_size, i+3, j+1);
175:       LINEARSOLVER (x, x0, a, inv_c, grid_size, i+3, j+2);
176:       LINEARSOLVER (x, x0, a, inv_c, grid_size, i+3, j+3);
177:     }
178:       }
```

CQA

Path 1 / 1 OK

Coverage 63.14 %
Function linearSolver1
Source file and lines kernel.c:15-176
Module hydro_k1
The loop is defined in /gpfs/home/nct00/nct00010/TESTS_HANDSON/MAQAO_HANDSON/hydro/kernel.c:15-176.

The related source loop is not unrolled or unrolled with no peel/tail loop.

gain potential hint expert

Vectorization

Your loop is not vectorized. Only 6% of vector register length is used (average across all SSE/AVX instructions).
By vectorizing your loop, you can lower the cost of an iteration from 41.50 to 2.59 cycles (16.00x speedup).

Details

All SSE/AVX instructions are used in scalar version (process only one data element in vector registers). Since your execution units are vector units, only a vectorized loop can use their full power.

Workaround

- Try another compiler or update/tune your current one:
  - use the vec-report option to understand why your loop was not vectorized. If "existence of vector dependences", try the IVDEP directive. If, using IVDEP, "vectorization possible but seems

CQA output for kernel1

```
> maqao oneview create-report=one xp=ov_k1 \
output-format=text text-cqa=138 | less
```

gain | potential | **hint** | expert

Type of elements and instruction set

80 SSE or AVX instructions are processing arithmetic or math operations on single precision FP elements in scalar mode (one at a time).

Matching between your loop (in the source code) and the binary loop

The binary loop is composed of 96 FP arithmetical operations:

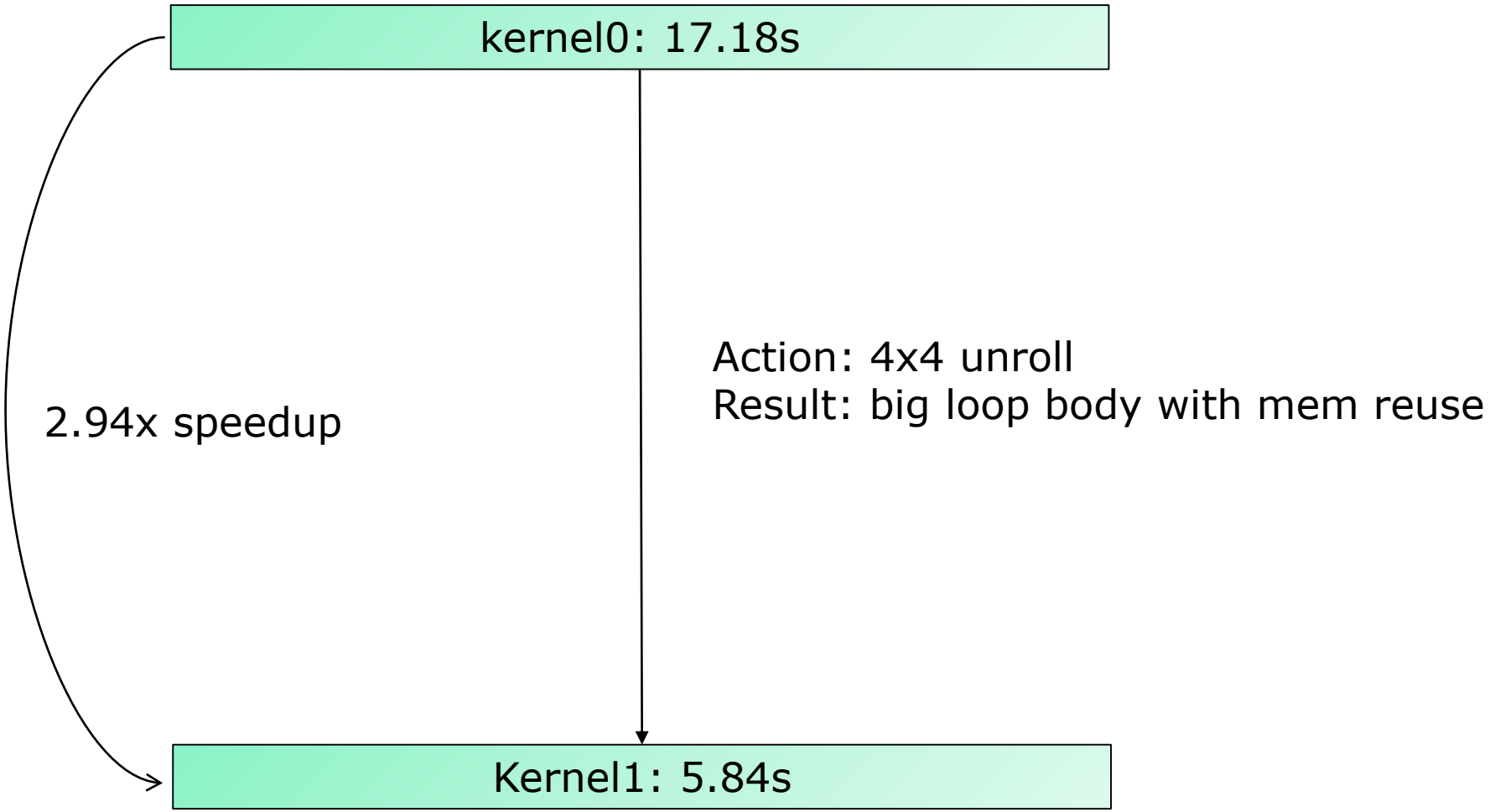
- 64: addition or subtraction
- 32: multiply

The binary loop is loading 272 bytes (68 single precision FP elements). The binary loop is storing 64 bytes (16 single precision FP elements).

4x4 Unrolling was applied

Expected 48... But still better than 80

Summary of optimizations and gains



Profiling with MAQAO ONE View 2 & 3

Cédric Valensi

Current issues and limitations with modes 2 & 3

- ONE View reports in mode 2 & 3 involve modules on which work is still ongoing
  - OpenMP is not fully supported by DECAN
  - Executing through job scheduler is not supported by VProf and DECAN
  - Since DECAN heavily modifies the binary, some of its transformations may crash
- Higher overhead than for report 1!
  - Instrumentation in binary loops can cause significant overhead
  - Multiple runs are performed
- Workarounds:
  - Execute in MPI only
  - Use interactive session to launch jobs
  - **Reduce number of analysed loops**

Setup ONE View for interactive run

- Retrieve the configuration file prepared for a NAS benchmark in batch mode from the MAQAO_HANDSON directory

```
> cd $HOME/NPB3.4-MZ/NPB3.4-MZ-MPI/
> cp $HOME/MAQAO_HANDSON/npb/config_maqao_npb_int.lua .
> less config_maqao_npb_int.lua
```

```
binary = "./bin/bt-mz.A.x"
...
number_processes = 4
...
number_nodes = 1
...
number_tasks_nodes = 4
...
mpi_command = "mpirun"
...
omp_num_threads = 1
...
filter = {
 type = "number",
 value = 15
}
```

Generate ONE View report two

- Request interactive session

```
> qsub -I -W group_list=PRACET1FWB_MAQAO -l walltime=0:30:00 \
-l select=1:ncpus=4:mpiprocs=4:mem=5gb
```

- Load environment

```
> module load compiler/intel/2018.0.1 intelmpi/5.0.3.049/64
> module load parallel_tools/maqao/2.8.6
```

- Generate ONE View report two

```
> cd $HOME/NPB3.4-MZ/NPB3.4-MZ-MPI/
> maqao oneview --create-report=two -xp=maqao_npb_int \
--config=config_maqao_npb_int.lua
```

- The HTML files are located in <exp-dir>/RESULTS/<binary>_**two**_html.

```
> firefox maqao_npb_int/RESULTS/bt-mz.A.x_two_html/index.html
```

- A sample result directory is in MAQAO_HANDSON/npb/bt-mz.A.x_**two**_html/

Generate ONE View report three

- Generate ONE View report three from previous directory

```
> cd $HOME/NPB3.4-MZ/NPB3.4-MZ-MPI/
> maqao oneview --create-report=three -xp=maqao_npb_int
```

- The HTML files are located in <exp-dir>/RESULTS/<binary>_**three**_html.

```
> firefox maqao_npb_int/RESULTS/bt-mz.A.x_three_html/index.html
```

- A sample result directory is in MAQAO_HANDSON/npb/bt-mz.A.x_**three**_html/

More sample codes

- More codes to study with MAQAO in

```
/projects/uvsq/PRACET1FWB_MAQAO/tutorial/loop_optim_tutorial.tgz
```

Additional examples

Using MAQAO on unsupported architectures

Cédric Valensi

MAQAO extended mode

- The modules composing MAQAO can be invoked separately
  - Useful to provide extended options
- LProf can profile applications on architectures not supported by MAQAO by relying on OS timers
- CQA can perform cross-analysis
  - Useful to predict behaviour on different architectures
  - Unsupported architectures can be analysed by specifying an architecture with a similar behaviour

- Retrieve jobscript modified for LProf from the MAQAO_HANDSON directory.

```
> cd $HOME/NPB3.4-MZ/NPB3.4-MZ-MPI/
> cp $HOME/MAQAO_HANDSON/npb/job_npb_lprof.pbs .
> less job_npb_lprof.pbs
```

```
...
mpirun ./bt-mz.C.x
<mpi_command> <run_command>
```

- Invoking Lprof with flag --use-OS-timers

```
> module load parallel_tools/maqao/2.8.6
> maqao lprof --use-OS-timers -xp=lprof_npb \
--mpi-command="mpirun" --batch-command="qsub" \
--batch-script="job_npb_lprof.pbs" -- bin/bt-mz.C.x
```

- Display LProf results

```
> maqao lprof -xp=lprof_npb -df    # Function profile
> maqao lprof -xp=lprof_npb -dl    # Loop profile
```

- Retrieve loop identifier(s) to analyse from LProf results

```
##########################################################################################
#  Loop ID   |    Module    |        Function Name        |         Source Info         |
##########################################################################################
#  208       |  bt-mz.C.x   |  matmul_sub                 |  solve_subs.f:71-175        |
#  186       |  bt-mz.C.x   |  x_solve                    |  x_solve.f:145-307          |
#  193       |  bt-mz.C.x   |  y_solve                    |  y_solve.f:144-306          |
#  214       |  bt-mz.C.x   |  z_solve#omp_region_0       |  z_solve.f:145-307          |
```

- Analyse loop(s) with CQA and specifying another architecture

```
> module load parallel_tools/maqao/2.8.6
> maqao cqa bin/bt-mz.C.x uarch=SANDY_BRIDGE loop=208
```